„Threaded Interpretive Languages“

Als *„Interpretive Languages“* werden in der Computerwelt *Interpretersprachen* (d.h. *interpretierende Programmiersprachen*) bezeichnet, also alle Programmiersprachen, deren Programmanweisungen über einen Befehlsinterpreter abgearbeitet werden.

Als *„Threaded Interpretive Languages“* werden Interpretersprachen bezeichnet, bei denen die interne Arbeitsweise des Befehlsinterpreters besonders elegant und durchdacht ist. Sie ziehen bei der Abarbeitung der Programmanweisungen intern eine Art Ariadnefaden hinter sich her, der sie bei der korrekten Abarbeitung der Programmanweisungen unterstützt. Der Begriff *„thread“* stammt aus dem Englischen und bedeutet so viel wie „einfädeln“ oder „aufreihen“. Mit „Threaded Interpretive Languages“ werden also genau die Interpretersprachen bezeichnet, deren Interpreter ein Ariadnefaden-Konzept verwenden. Die Programmiersprache *Forth* dürfte zumindest dem Namen nach die bekannteste dieser Sprachen sein.

Da Forth zudem eine sehr einfache Programmiersprache ist, ist es sinnvoll, die Arbeitsweise von „Threaded Interpretive Languages“ anhand einfacher Forth-Beispielen vorzustellen. Eine Einführung in Forth liefert in dieser Abhandlung das Material, um die Arbeitsweise einer „Threaded Interpretive Language“ kennenzulernen.

1. Einführung

Die Anforderungen an die Software von Mikrocomputern und Minirechnern sind in den letzten Jahren stark gestiegen. Komplexere und leistungsfähigere Hardware, aber auch anspruchsvollere Problemstellungen bei gestiegenem Basis-Know-How sorgen dafür, daß sich hier eine Entwicklung wiederholt, die von den Großrechnern her hinlänglich bekannt ist. Für die Bewältigung der hiermit einhergehenden Probleme wird mit der Computersprache **Forth** eine moderne, insbesondere für systemnahe oder zeitkritische Anwendungen geeignete Lösung dargestellt.

Der Einsatz eines Computers erfolgt im wesentlichen immer in den folgenden Schritten:

1. Definition der Aufgabe, die der Rechner zu erledigen hat, mit eindeutiger Spezifikation aller Eingangsdaten und aller Ausgangsdaten (bzw. Resultate).
2. Entwurf eines Algorithmus oder einer Folge von Schritten, mit denen der Rechner diese Zuordnung erzielen kann.
3. Formulierung dieses Algorithmus in einer geeigneten, dem Rechner verständlichen Form.
4. Austesten des Algorithmus (und, falls dies nicht auf dem Zielrechner erfolgt, anschließend Zielrechner-Implementierung und Austesten auf dem Zielrechner).
5. Programmpflege, d.h. fallweise Anpassung an veränderte Anforderungen oder Korrektur nachträglich entdeckter Fehler usw.

Gestaltung und Ablauf dieser Phasen hängen wesentlich von Art und Umfang eines Projektes ab.

Was ist Forth?

Mehr als bei jeder anderen Computersprache hängt die Antwort darauf vom Standpunkt des Benutzers ab. Forth kann

- als eine Sprache angesehen werden,
- aber auch als Betriebssystem,
- als Interpreter
- ebenso wie als interaktiver Compiler,
- als erweiterbare Datenstruktur
- wie auch als Softwarekonzept,
- und es ist auch ein polarisierendes Konzept.

Forth ist <u>sehr verschieden</u> von mehr konventionellen Sprachen wie *BASIC, ALGOL, PASCAL* oder *C*. Einige seiner Strukturen haben kaum Entsprechungen in diesen Sprachen, sodaß Forth eine Programmierumgebung mit

- außerordentlichen Stärken,
- Werkzeugen,
- aber auch stilistischen Besonderheiten

erzeugt.

Da die Sprache ursprünglich für Echtzeit-Anwendungen (real-time-Anwendungen) entwickelt wurde, weist sie Eigenschaften auf, die sie ideal geeignet macht für

- Meßdatenerfassung,
- Prozess- und Maschinensteuerungen,
- automatische Testsysteme,
- Automation und vergleichbare Bereiche.

Das Ergebnis ist eine einzigartige Sprache, die strukturiertes Programmieren unterstützt und modulare Programme erzeugt, deren Interaktivität und Flexibilität das Debuging erleichtert und dazu noch die Programmfehlerhäufigkeit verringert und die Zuverlässigkeit der erzeugten Software steigert. Die so erzeugten Programme sind leicht und zuverlässig zu aktualisieren, haben sehr gute Laufzeiteigenschaften und führen zu kompaktem Code.

Forth ist erweiterbar. Das bedeutet, daß der Benutzer der Sprache eigene Befehle hinzufügen kann. Neue Worte werden definiert unter Verwendung älterer Worte oder unter Rückgriff auf die Maschinen- bzw. Assemblersprache des gewählten Prozessors. Dies erfolgt im Prinzip so lange, bis ein einziges Wort das gesamte gewünschte Programm darstellt. Wird dieses Wort, das das gewünsche Programm darstellt, dann in Forth aufgerufen, so wird das Programm zur Ausführung gebracht. Dieses Wort ist also der in Forth hinterlege Name (Forth-Name, Wortname) des Programms. In Forth-Büchern findet sich dazu häufig der Satz: *„In Forth wird ein Programm durch Aufruf seines Namens zur Ausführung gebracht."*

Duch dieses Vorgehen kann ein Forth so erweitert werden, daß es zu einer Sprache heranreift, die einen Grade an Qualität und Einfachheit erreicht hat, der ihre Anwendung auch für Nicht-Programmierer möglich macht.

Der Name „Forth“ ist in typisch amerikanischer Art als Schlagwort entstanden und meint „Sprache für die vierte Generation“, wobei der Begriff Generation nicht sonderlich eng aufzufassen ist.[1] Die Sprache *Forth* ist auf Maschinen der verschiedensten Größe und Art lauffähig, wobei natürlich (wie es einer modernen Sprache gemäß ist) von modernen Strukturen Gebrauch gemacht wird, die in manchen kleineren bzw. älteren Maschinen zu emulieren sind. In diesem Sinne ist der Name aber auch als Imperativ zu verstehen im Sinne einer vorwärtsgerichteten, zukunftsorientierten Sprache (mit dem gleichnamigen Fluß in Schottland bestehen dagegen keinerlei Zusammenhänge). Sie wurde in den 1960er Jahren als 1-Mann-Projekt von ***Charles „Chuck“ H. Moore*** entwickelt und schließlich 1968 erstmals von ihm publiziert.

Warum überhaupt Progammiersprachen?

Die Arbeitsgeschwindigkeit von in *Assembler* geschriebenen Programmen ist in vielen Anwendungen erforderlich, dagegen ist es oft wünschenswert, die *Produktivität* bei der Programmerstellung und auch die *Zuverlässigkeit* und *Lebensdauer* von Programmen zu erhöhen, indem man eine *High-Level-Sprache* verwendet (in keinem dieser Punkte steht *Assemblerprogrammierung* bei größeren Projekten sonderlich gut da). Forth ist entworfen worden mit Blick darauf, diesen widersprüchlichen Forderungen gerecht zu werden.

Für die Realisierung kleinerer Softwareprojekte ist insbesondere bei extrem zeitkritischen Problemstellungen die Programmierung in *Maschinensprache* bzw. *Assembler* adäquat. Allerdings zeigt sich mit steigender Komplexität einer Aufgabe recht bald und immer stärker die Notwendigkeit, strukturiert und modular zu programmieren, Softwaremodule zu integrieren und das Projekt durch Trennung von *Betriebssystem* und *Applikation* übersichtlicher zu halten. Zwischen Softwareblöcken werden *Parameterübergabe-Prozeduren* nötig und bei Aufruf der Funktionen werden *Regeln* zu beachten sein, sodaß sich allmählich der gesamte Zeit- und Programmoverhead dem bestimmter Programmiersprachen nähert, ohne deren Vorteile zu nutzen. Die Wahrscheinlichkeit unentdeckter Fehler steigt. Ein Beispiel soll die Situation veranschaulichen.

2. Parameter Passing

Angenommen sei beispielsweise ein größeres Assemblerprogramm, in dem die Parameterübergabe „standardisiert“ über Literale im Anschluß an den jeweiligen Funktionsaufruf (Subroutine) erfolgen soll. Dabei können die Literale Pointer zu den eigentlichen Parametern sein oder (bei geringerer Flexibilität) auch direkt benutzt werden. Dann hat dieses Programm etwa folgende Gestalt:

```
        JSR     ...
        LDAA    ...
        JSR     FUNKTION1        ; z.B. Division zweier Variabler
LABEL1  DC      A(LITERAL11)     ; Übergabe des Zählers direkt oder indirekt
LABEL2  DC      A(LITERAL12)     ; Übergabe des Nenners in gleicher Weise
LABEL3  .       ...              ; die Labels sollen nur der besseren
        ASL     ...              ; Übersicht dieses Beispieles dienen
```

Damit ist offensichtlich eine durchaus überschaubare Gestalt des Hauptprogrammes möglich. Allerdings muß der Zugriff auf die Parameter erst durch ein Hilfsprogramm ermöglicht werden, da kein Prozessor dies unterstützt (er würde nach Erledigung der `FUNKTION1` „normalerweise“ nach `LABEL1` zurückkehren!). Nehmen wir ferner an, daß dieses Hilfsprogramm als Subroutine gleich zu Beginn in der `FUNKTION1` aufgerufen wird, so werden zwei Rückkehradressen auf dem Stack liegen. Die „ältere“

dieser Adressen erlaubt über eine indirekte Adressierungstechnik den Zugriff auf die Literale. Anschließend muß diese Adresse auf dem Stack mittels Offsetrechnung auf LABEL3 adjustiert werden.

Falls aufwendigere Stackmanipulationen vermieden werden sollen, kommt für die Parameter nur eine Ablage in Registern oder RAM-Hilfszellen in Betracht. Falls man rekursiv programmieren möchte (oder reentrant bei Multitasking oder auch bei Interrupt-Handling), wird Rettung nötig. Wohin? Falls auf den Stack (oder gar auf einen zusätzlichen Parameterstack), so wäre die Übergabe von vorn herein auf dem Stack günstiger, obwohl dann Parameter und Adressen je nach Komplexität bunt gemischt sein können (was das Beispiel indirekt allerdings auch zeigt). Forth zieht hier eine rigorose Konsequenz, die letztlich zu erstaunlich einfachen und leistungsfähigen Programmstrukturen führt.

Forth arbeitet mit zwei Stacks: *Parameterstack* und *Returnstack*. Die Parameterübergabe an Funktionen erfolgt zumeist im Sinne lokaler Variabler auf dem Parameterstack, wohingegen der Returnstack normalerweise für Kontrollstrukturen und Returnadressen reserviert ist . Literale treten nur in Form compilierter Konstanten auf. Daneben stehen natürlich Elemente (Datentypen) wie VARIABLE oder CONSTANT und frei definierbare Datenstrukturen zur Verfügung.

Postfix-Notation

Das Beispiel zeigt eine allgemeine Besonderheit bei der maschinennahen Parameterübergabe an Softwaremodule: *Funktionsaufruf und Parametertransfer erfolgen natürlicherweise getrennt*. Bei der Literal-Handler-Methode z.B. wird zuerst die Funktion gerufen und anschließend die zugehörigen Werte (Präfix-Notation). Allerdings kehrt sich diese Reihenfolge in Wahrheit um, wenn man den Zeitpunkt betrachtet, zu dem die Funktion vor ihrer eigentlichen Ausführung auf die Parameter zugreift. In Forth wird eine Funktion ihre Parameter auf dem Stack erwarten, d.h.

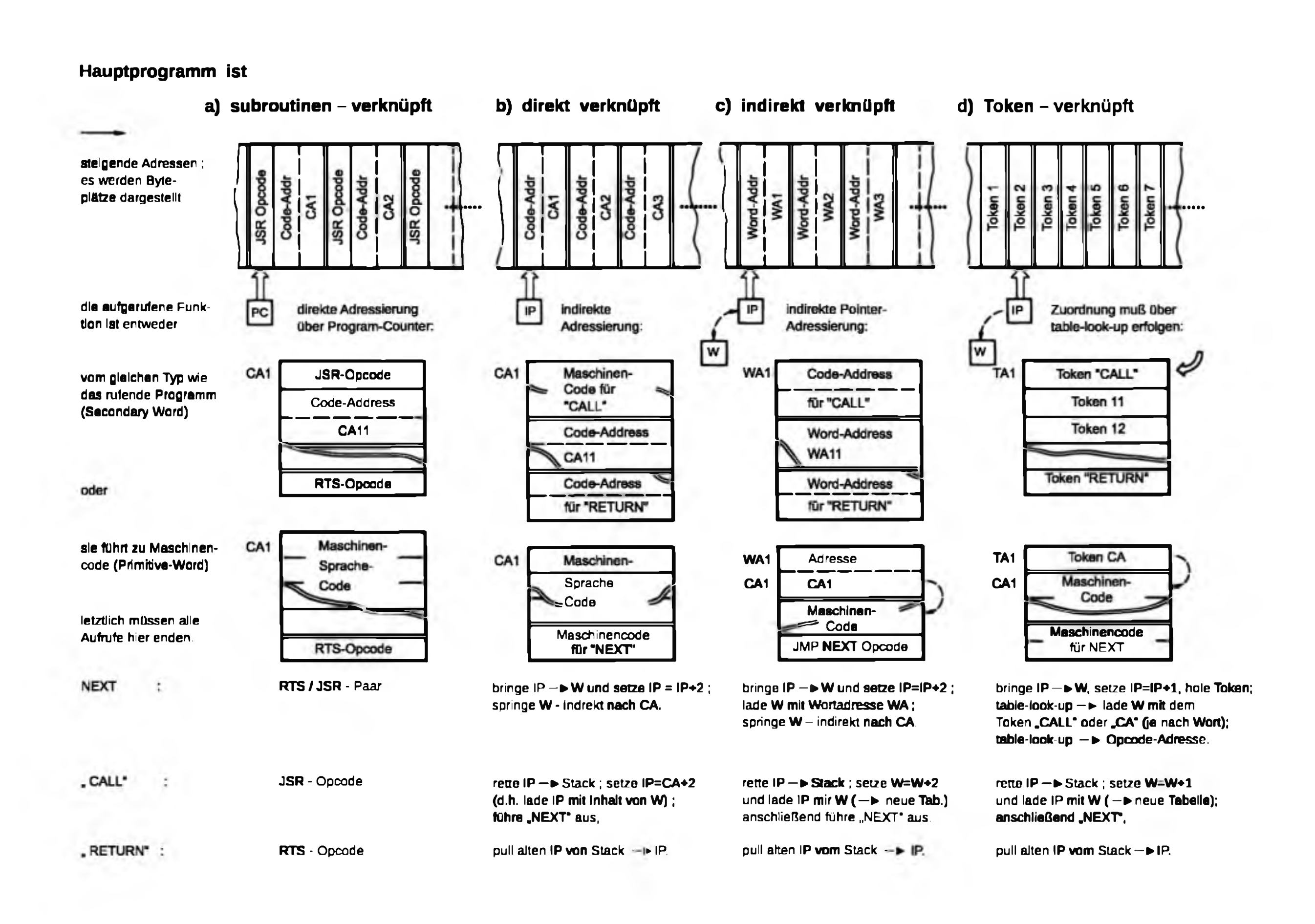
> **Forth arbeitet mit *Postfix-Notation***, auch als *„Umgekehrte Polnische Notation“ (UPN)* bekannt.[2] Es sei an einen weit verbreiteten wissenschaftlichen HP-Taschenrechner erinnert. Die Übernahme dieses Konzeptes in eine High-Level-Sprache ist ein wesentlicher Grund für Laufzeit- und Codeeffizienz, Einfachheit und Zuverlässigkeit dieser kompakten Sprache.

Die Einarbeitung in Forth fällt Assembler-Programmierern, aber auch Novizen bisweilen leichter als erfahrenen Programmierern anderer High-Level-Sprachen, insbesondere wenn es sich hierbei um Sprachen wie *BASIC*, *FORTRAN* oder *PL/1* handelt (nicht so sehr im Falle der *C*-Sprache). In Forth erwarten die Funktionen (Worte) ihre Argumente in der Mehrzahl der Fälle als ‚blanke Zahlen‘, statt als Referenzen zu Variablen oder Konstanten. Selbst dort, wo dies in Forth nach außen nicht direkt in Erscheinung tritt, weil zum Beispiel (relativ selten) dennoch mit solchen Referenzen gearbeitet wird, ist trotzdem zumeist ein Stack funktional ein tragendes Element. Das ist gleichbedeutend mit der Aussage, daß eine Funktion ihre Argumente zumeist in Form temporärer (lokaler) Variablen erhält und auch wieder übergibt. Allerdings liegt es natürlich völlig in der Hand des Anwenders, sich hier seine ganz persönliche Programmierumgebung zu schaffen und aus den Möglichkeiten zu lokalem (Stack-) Parameter-Passing und zu globalem Parameter-Passing für jede konkrete Anwendung den größtmöglichen Vorteil zu ziehen.

Das Arbeiten mit Stacks ist jedoch dermaßen einfach und vorteilhaft, daß von daher eine natürliche Tendenz in Forth besteht, globale Größen (wie Variable, Konstante, usw.) nur sparsam zu verwenden.

3. Verknüpfung von Softwaremodulen

Die Zusammenfügung unterschiedlicher (möglichst in sich abgeschlossener) Softwarebausteine zu einem einheitlichen Programm ist ein weiteres zentrales Problem, das großen Programmen und Sprachen gemeinsam ist. Es sollen kurz vier für unser Thema bedeutsame Verfahren vorgestellt werden, die in der Literatur unter dem Sammelnamen „**Threaded Code Structures**" oder „**Threaded Codes**" bekannt sind. *Abb. 1* zeigt diese Strukturen schematisch.



– *Abb. 1: Verknüpfungsstrukturen* –

Die Abbildung 1 ist wie folgt zu lesen:

Die erste Bildzeile zeigt symbolisch ein Programm oder ein ‚Hauptprogramm' auf Maschinen-Ebene in seiner physikalischen Repräsentation. Dessen Byte-Plätze sind konsistent angedeutet, wobei aber fallweise die Gruppierung von Bytes in Adreßworte (‚Zellen') vorgenommen wurde.

Diese Programmstruktur wird unter Zuhilfenahme eines Zeichers (‚Pointers') abgearbeitet, wobei dieser Pointer auf den jeweils aufzurufenden Befehl oder auch auf sein Inline-Argument (z.B. die Code-Adresse für einen Jump-Subroutine-Opcode im Falle a) in Abb. 1) zeigt. Dieser Pointer kann ein spezialisiertes Maschinenregister (‚Program Counter' - PC) sein oder auch ein generalisiertes Register oder Software-Register IP (‚Instruction Pointer'). Mit dem Aufruf eines Elementes geht ein passendes Inkrement dieses

Pointers einher, damit nach Abarbeitung des aktuellen Befehles auf das folgende Element zugegriffen werden kann.

Die beiden folgenden Bildzeilen zeigen symbolisch in gleicher Weise Programme, nun aber solche, mit denen die gewünschten Befehle realisiert werden. Dabei ist zu unterscheiden, ob das so aufgerufene Programm noch von gleicher Gestalt ist wie das rufende Programm (Aufruf eines ‚High-Level-Codes'), oder ob es bereits aus konkret ausführbarem Code besteht (Aufruf von Maschinen-Code).

Im Falle eines High-Level-Aufrufes muß man sich die Abbildung entsprechend fortgesetzt vorstellen. Letztlich muß eine Aufrufkette natürlich immer mit dem Aufruf von Maschinencode enden. Dessen Startadresse CA (‚Code Adress') muß also in letzter Konsequenz gefunden werden (und auch der Low-Level-Instruction-Pointer PC muß letzlich auf diese Startposition gesetzt werden). Auch die höchste Hochsprache kommt schließlich nicht ohne eine zugrunde liegende Maschine (Mikroprocessor, Mikrocontroller) aus.

Sofern diese Code-Adresse nicht in direkter Weise im Hauptprogramm gefunden wird, sondern dort über einen Pointer definiert ist, spricht man von ‚**indirekter Verknüpfung**'. Die Adresse, in der dieser Pointer zu finden ist, wird mit *W*ort-*A*dresse WA oder auch häufig mit CFA (‚Code Field Address') bezeichnet.

Im Falle einer symbolischen Repräsentation (Abb. 1, d)) wird diese Zeile mit TA (‚Token Address') bezeichnet.

Der Fall a) bedarf keiner Erklärung. Er stellt den denkbar einfachsten Fall dar, wenngleich den einzigen, der von allen modernen Prozessoren voll unterstützt wird. Wegen offensichtlicher Optimierungsmöglichkeiten ist diese Struktur nur als Resultat eines Compilationsprozesses im Zusammenhang mit Programmiersprachen bedeutsam.

Es hat sich gezeigt, daß die konventionell von CPUs (Mikroprossoren, Mikrocontroller) angebotenen Kontrollelemente zur Realisierung mächtiger Sprachstrukturen nicht ausreichen. Die weiteren in Abb. 1 gezeigten Strukturen arbeiten (in Reihenfolge b), c), d)) mit zunehmend komplexeren indirekten Adressierungstechniken, und allen drei Versionen gemeinsam ist ein ‚Instruction-Pointer' (IP), der auf Sprachelementebene ähnliche Kontrollfunktionen ermöglicht wie der ‚Program-Counter' (PC) auf der Maschinenebene.

Abb. 1 stellt natürlich Archetypen dar. So würde das Strukturbeispiel in Abschnitt 1.1., wenn man außer JSR-Befehlen keine Inline-Befehle im Hauptprogramm zuließe, dennoch der Abb. 1a) nicht voll entsprechen: das Literal-Handling erfordert fallweise die Manipulation einer Indirektadresse, die jedoch nur (auf dem Stack) flüchtig existiert. Das Beispiel zeigt also eine Mischform der Strukturen nach 1a), 1b), die in umfangreichen Assemblerprogrammen durchaus üblich ist.

Die Wahl einer Verknüpfungsstruktur ist von vielen Umständen abhängig. So scheint auf den ersten Blick der Overhead von 1b) kleiner als der von 1c) zu sein. Jedoch muß <u>ein direkt aufgerufenes Wort</u> (*direkte Verknüpfung, direkte Adreßierung*) immer mit Maschinencode starten, und falls das Wort vom gleichen Typ wie das rufende Programm ist (*High Level Definition*), so muß eine Call-Prozedur von diesem Code durchgeführt werden. D.h. verschiedene Worte haben verschiedene Entries; der Overhead steckt großenteils in den einzelnen Worten.

Ein <u>indirekt verknüpftes Programm</u> (*indirekte Verknüpfung, indirekte Adreßierung*) ruft dagegen sämtliche existierenden Worttypen (Abb. 1 ist hierin natürlich nicht vollständig(!)) in einheitlicher Weise auf. Mittels indirekter Adressierung über die aktuelle Wort-Adresse (WA) im momentan sbzuarbeiteten Hauptprogramm wird in der Wortadresse WA des aufgerufenen Wortes eine Code-Adresse (d.h. ein

Adreßpointer zu „ausführbarem Code“) gefunden, die den Typ des aufgerufenen Wortes definiert. Die WA führt also immer zu einer CA, durch die wiederum *über Maschinencode* der Wort- oder Datentyp festgelegt ist, indem sie entweder zu ausführbarem Code oder zu einer neuen Wortadreßliste vom Typ des aufrufenden Programmes führt. Beliebige Datentypen und Programmstrukturen sind in High-Level (d.h. auf Wortebene) oder in Low-Level (Maschinenebene) definierbar und werden ohne Ausnahme einheitlich behandelt.

Vorteile dieses Verfahrens sind

- einfacher Wortaufbau,
- einheitliche Entrysituation und
- damit einfache Compilerstruktur (bei Generierung über eine Sprache).

Der besondere Wert einer indirekt verknüpften Struktur wird sich allerdings erst bei der Behandlung komplexerer ‚Defining-Words‘ (definierende Wörter) wie die Forth-Wörter `;CODE` oder `DOES>`. Minirechner-CPUs und in zunehmendem Maße auch moderne Mikroprozessoren unterstützen diese Indirekt-Aufruf-Prozeduren inklusive Pointer-Manipulationen in steigendem Maße direkt, sodaß zunehmend auch von dieser Seite her indirekt verknüpfte Strukturen begünstigt werden. Im Hinblick darauf, daß bei den Strukturen nach Abb. 1 die Notwendigkeit besteht, Adressen zu „interpretieren“, hat sich die Bezeichnungsweise „***threaded interpretive language codes***“ eingebürgert. Der Ausdruck „*thread*“ (engl., einfädeln, aufreihen) ist am besten so zu verstehen, daß bei verschachtelten Aufrufen mit beliebiger Tiefe immer ein „Ariadnefaden“ gezogen wird (Rückkehradreß-Kette), an dem entlang zur rufenden Ebene zurückgekehrt wird. Er kann aber genau so gut auf die Art und Weise bezogen werden, wie Programme („Worte“) in sog. ‚*Dictionaries*‘ (Wörterbüchern, Wortlisten) untereinander verknüpft sind. Die mit der Aufrufprozedur befaßten Programmteile werden „***innerer Interpreter***“ genannt. Eindeutig hiervon zu unterscheiden ist der „***äußere Interpreter***“, der hier nicht dargestellt ist, der aber als Teil des Mensch-Maschine-Interface bei einer High-Level-Programmiersprache die Eigenschaften der Sprache festlegt und Codes der dargestellten Art generiert. Abb. 1d) zeigt eine mögliche token-verknüpfte Struktur.[3] Legt man die Hauptprogramm-Token (vom Äußeren Interpreter gestützt) in numerierten Token-Files ab und erweitert die Struktur mit High-Level-Literal-Handlern und Pointer zu den Lines (Programmzeilen) und innerhalb der Lines, so kommt man zu *BASIC*-ähnlichen Strukturen. Traditionell ist in diesen Strukturen keine klare Trennung zwischen dem äußeren und dem inneren Interpreter mehr vorhanden. Sie arbeiten bei Eingabe und bei Ausführung eines Programmes im Wortsinne interpretativ und entsprechend langsam.

4. Forth sui generis

Im Gegensatz hierzu ist bei der High-Level-Sprache Forth eine sehr scharfe Trennung zwischen dem *Inner-Interpreter* (*Innerer Interpreter*) und dem *Outer-Interpreter* (*Äußere Interpreter*) realisiert. **Forth ist eine indirekt verknüpfte Sprache**, vgl. Abb. 1c). Der Äußere Interpreter arbeitet interaktiv und interpretativ, basierend auf einer *Threaded-List-Structure*, die von den sog. Headern der Forth-Standardworte und den durch Programmiertätigkeit hinzugefügen (d.h. neu definierten) Erweiterungen (Programmen) gemeinsam gebildet wird.

Eines der vielen Worte, die der Outer Interpreter[5] auf diese Weise entdecken könnte, ist das Wort „**:**“ \,[6] (das Doppelpunkt-Zeichen). Wird dieses „Wort“ eingegeben,so schaltet der Äußere Interpreter um in den

sog. *CompIle Mode*, d.h. ab da wird ein neues Wort definiert. Es wird mit einem Header versehen, mit dem es später aufgerufen werden kann; nachfolgend über die Tastatur eingegebene Worte werden mit ihren WAs in das neue Wort compiliert. Zurückschaltung in den interpretiven Betriebszustand erfolgt mit „**;**" (das Strichpunkt-Zeichen (Semikolon)). Ruft man nun dies neue Wort auf, so findet es der Äußere Interpreter im Dictionary per Suchlauf und läßt es anschließend nach Übergabe der WA an den Inneren Interpreter ausführen (vgl. Abb. 1c) und ff.). Statt dessen könnte das neue Wort auch Bestandteil einer noch neuen Definition werden usw.
Ein Beispiel sagt jedoch mehr als tausend Worte.

Zuvor müssen wir allerdings einige Vereinbarungen zur Schreibweise treffen, da die spezielle Notation in Forth leicht zu Verwechslungen mit normalem Text in einer Abhandlung Buche führen kann. Um dies zu vermeiden, werden Forth-Ausdrücke an kritischen Stellen in Fettdruck hervorgehoben. Wo zur Veranschaulichung nötig, werden die üblichen Trenn- und Abschlußzeichen „**<space>**" und „**<return>**" mit den Symbolen „␣" und „⏎" markiert (Forth-Wörter müssen mit mindestens einem „**<space>**" voneinander getrennt werden bzw. mit einem „**<return>**" abgeschlossen werden). Wo nötig, wird zudem die Antwort des Rechners unterstrichen, um sie von Tastatureingaben oder begleitendem Buchtext zu unterscheiden.

Angenommen, wir wollen in einem Behälter zwei Granulate mischen und möchten dazu die Klappen an den Einfüllstutzen vom Computer aus steuern. Die Definitionen

```
    :␣SCHIEBER.1.auf␣ ... ␣;
    :␣SCHIEBER.1.zu␣  ... ␣;     sowie entsprechende Definitionen für Schieber 2
und :␣SEKUNDEN␣       ... ␣;
```

mögen bereits existieren. Die drei Pünktchen („...") deuten dabei Worte an, die bereits in High-Level (d.h. unter Verwendung „älterer" Worte) oder in Maschinencode definiert sein können.

Angenommen sei ferner, daß 1 kg Granulat 1 in drei Sekunden bzw. 1 kg des Granulat 2 in fünf Sekunden einfließen, sobald die Klappe geöffnet ist. Das Wort **SEKUNDEN** möge so definiert sein, daß es eine Zahl auf dem (Parameter-)Stack erwartet und eine Zeitschleife mit der entsprechenden Dauer abarbeitet.

Sodann können wir z.B. definieren:

```
:␣kg.GRANULAT.1␣SCHIEBER.1.auf␣3␣*␣SEKUNDEN␣SCHIEBER.1.zu␣; ⏎
:␣kg.GRANULAT.2␣SCHIEBER.2.auf␣3␣*␣SEKUNDEN␣SCHIEBER.2.zu␣; ⏎
```

Wenn wir jetzt eingeben

```
4␣kg.GRANULAT.1␣␣6␣kg.GRANULAT.2 ⏎ OK
```

so werden 10 kg Granulatgemisch im Mischverhältnis 40% von Granulat 1 zu 60% von Granulat 2 in dem Kasten landen und der Steuerungscomputer gibt ein „OK" (für okay, dies ist die Standardantwort eines interaktiven Forth-Systems nach fehlerfreier Ausführung) auf dem Bildschirm aus.

Zu beachten ist die typische Übergabe der Parameter *vor* den Operanden oder Definitionen. Beispielsweise erwartet die Definition (Wort) **kg.GRANULAT. 1** auf dem (Parameter-) Stack, der in

Zukunft nur noch kurz „*Stack*“ genannt wird, eine Zahl. Auf der Ebene dieser Definition ergibt sich also eine sehr natürliche Schreibweise.

Innerhalb des Wortes aber ergibt sich die Postfixschreibweise an der Stelle, wo aus der kg-Angabe und der Proportionalkonstante (die 3) die adäquate Schieberöffnungszeit errechnet wird:

```
... 4 3 * ...
```

An dieser Stelle soll hervorgehoben werden, daß typische Forth-Programme streng modular aufgebaut werden können (und sollen) und daß sie in modular komplett (!) austestbarer Weise formuliert werden können. So können auf jeder Ebene einer Programmentwicklung Module benutzt werden, deren Fehlerfreiheit bewiesen ist. Darüber hinaus zeigt das Beispiel die Möglichkeit, Programme in sehr gut lesbarer Gestalt zu formulieren (und zu dokumentieren). Dies verdient auch deshalb hervorgehoben zu werden, weil Forth nicht ganz zu Unrecht im Rufe steht, eine „write-only-Sprache“ zu sein, denn es braucht ein wenig Disziplin, um schön lesbare Forth-Programme zu schreiben.

Bemerkenswert ist die Entstehungsweise und der innere Zusammenhang komplexer Definitionen. Im obigen Beispiel ist für die konkrete Gestaltung ausschlaggebend, daß die endgültige Kommunikation mit dem Computer in einer natürlichen benutzerorientierten Weise abgewickelt werden soll. Schreiben wir uns also auf dem Papier hin, wie wir

- *vier kg Granulat 1*

und

- *sechs kg Granulat 2*

anfordern wollen, so sehen wir unmittelbar, wie die Definition

- `kg.GRANULAT.1`

bzw. ihr Gegenstück

- `kg.GRANULAT.2`

etwa aussehen sollten. Hieraus ersehen wir wiederum die Notwendigkeit, noch grundlegendere Definitionen wie

- `SCHIEBER.1.auf`
- `SCHIEBER.2.zu`

oder auch

- `SEKUNDEN`

aufzubauen, von denen einmal angenommen werden soll, daß sie endlich elementar in Forth-Standardworten oder in Maschinencode (bzw. Assembler) formulierbar sind.

Nach dieser „*Top-Down-Entwurfsprozedur*“ wird dann das Programm in der dargestellten Weise in „*Bottom-Up-Manier*“ endgültig in das Forth-System eingetippt und abgespeichert.

Forth führt also ganz natürlich zu *modularer Programmierung*, zu *modularer Austestbarkeit* und damit zu *einfachem Management* bei größeren Softwareprojekten (inklusive Entwurfsphase und Programmpflege) und infolgedessen zu bemerkenswert zuverlässiger Software(!).[7]

Die Sprache ist strukturiert (d.h. eine GOTO-Anweisung ist unnötig und wird auch nicht angeboten; alle Definitionen sind abgeschlossen, usw.) und weist die High-Level-Strukturelemente üblicher High-Level-Sprachen auf wie `DO...LOOP` , `IF...ELSE...ENDIF` , `BEGIN...UNTIL` usw. Sie vereinigt in sich die besten Dinge zweier Welten, indem sie darüber hinaus nicht nur mit kleinem Zeit- und Codeoverhead arbeitet,[8] sondern extrem einfache Möglichkeiten zur Einbindung von Maschinensprache- bzw. Assemblerprogrammen bietet.

Für kleine Maschinensprachedefinitionen können diese einfach direkt in Forth-Definitionen eingetragen werden, während bei größeren Maschinenprogrammen entweder extern assembliert und sodann in Forth eingebunden wird oder aber auch ein Assembler in Forth realisiert bzw. genutzt werden kann, der als 1-Pass-Assembler direkt in den Forth-Worten strukturierten Maschinencode aufbaut. Forth bietet jedoch noch mehr. Wenngleich die interaktive Arbeitsweise im Zusammenhang mit interpretativem und compilierendem Betrieb bereits genannt wurde, kann ihre Bedeutung beim Austesten von Programmen z.B. in Zielrechnerapplikationen (d.h. Austesten direkt im Zielrechner ohne die Notwendigkeit beispielsweise zu Simulation oder Emulation) nicht genug unterstrichen werden. Programme können in größtmöglicher Maschinennähe entstehen, und zwar im doppelten Sinn des Wortes.

Darüber hinaus ist Forth, wie eingangs gesagt, nicht nur eine Sprache, sondern sie enthält ein komplettes Betriebssystem mit Terminal-IO-Handlern und Disk- (oder Band-) *Operating System*, das über die Sprache vollständig zugänglich ist, d.h. **Forth ist auch ein komplettes Betriebssystem.** Damit ist Forth ein ideales Softwarewerkzeug, das dem Benutzer bei systemnaher Programmierung die totale Kontrolle über seine Maschine erlaubt. Hier liegt eine weitere große Stärke von Forth im Verhältnis zu anderen Sprachen. Andererseits wird die Sprache in Form von FIG-Forth als kleine und kompakte Sprache mit typisch 5 bis 7 kByte publiziert. Zudem finden Sie auf der Webseite der *Forth Interest Group* (*fig*) weitere freie Forth-Implementierungen aufgelistet.

Forth ist für buchstäblich alle Zwecke ausbaubar. Dennoch wird mancher Anwender gerade seine spezielle Applikation stiefmütterlich behandelt finden, jedoch nicht bereit sein, den nötigen Ausbau zu einer (seiner!) Spezialsprache vorzunehmen. Dieser Anwender hat aber auch normalerweise keinen Zielrechner (System), Echtzeit-(Interrupt-)Handling, ein zu automatisierendes Labor oder eine Fabrikation, eine Klinik, ein Forschungsinstitut, ein großes Radiotelekop samt Auswertungseineiten Datenenbank zu programmieren und zu steuern, sondern einfach ein einzelnes (irgendwie numerisch zu behandelndes) Problem, und er mag bei der Programmiersprache *BASIC* bleiben.

So enthält der *fig-Forth-Standard* beispielsweise keine *Floating-Point-Arithmetik*, da diese für viele Applikationen überflüssig wäre. Der Benutzer kann sie jedoch mit wenigen hundert Byte an ergänzenden Definitionen hinzufügen, wenn er sie braucht. Praktisch läßt sich jede Art von fremden Programmen in Forth einbinden, so z.B. auch Program-Packages oder (falls Forth in einem Host läuft) auch Host-Subroutinen.

Ebenso ist in Forth ein Assembler meist nur optional, da man sich recht häufig auch ohne behelfen kann. Er kann aber einfach hinzugeladen werden, und dies gilt auch für komfortable Disk-Editier-Möglichkeiten und viele andere Dinge.

Nach dem eben gesagten liegt für den Leser vielleicht der direkte Vergleich mit *Tiny Basic* o.ä. nahe, der aber nicht wirklich (und nicht einmal bei Ausklammerung der Betriebssystemeigenschaften von Forth)

trifft. Infolge der erweiterbaren Struktur, der Compilereigenschaften und des kleinen Overheades ist Forth potentiell eine große Sprache. Aber auch der immer noch skeptische Leser, der vielleicht doch lieber auf *PASCAL* zurückgreifen möchte, wird von Forth nicht in Stich gelassen: Forth ist bestens als *META-Sprache* zu benutzen.[9] Forth ist ebenso portabel wie der sonst übliche P-Code für *PASCAL*, und ein so gestalteter Rechner ist zweisprachig.

Forth ist eine lebende Sprache. So bleibt es nicht aus, daß es große und kleine, nah und fern verwandte Versionen auf dem Markt gibt (*URTH*, *ZIP*, *STOIC*, *CONVERS*, *MMSForth* u.v.a.). Manche sind für bestimmte Maschinen geschrieben (wie *MMSForth* für *TRS80* oder *CONVERS* für *DIGITAL*-Maschinen), andere für bestimmte Prozessoren optimiert wie *ZIP* für den *Z80*.[10] In diesem Reigen stellt ***FIG-Forth*** so etwas wie einen Standard dar, das es für eine weite Zahl unterschiedlichster Computer und Prozessoren vorliegt, als *Public Domain* veröffentlich und vertrieben wird, vollständig dokumentiert ist und komplett im Quelltext als *Public Domain* verfügbar ist – eine kollaborative Arbeistleistung der ***Forth Interest Group*** *(fig)*.

1.4. Forth-Standards

Forth-Standards, die auf den *fig-Forth-Standard* zurückgehen sind:

- *fig-Forth*-Standard
- *Forth-79*-Standard
- *Forth-83*-Standard
- *ANS-Forth*-Standard
- *Forth-200x*-Standard
- Aktueller *Forth-Standard* (und dessen Weiterentwickung)

---

1 Die Sprache erblickte die Bits der Welt auf einer *IBM 1130*, einem sog. Rechner der dritten Generation. Charles Moore, der Erfinder von Forth, berichtet, daß die Bezeichnung „FORTH“ anstelle von „FOURTH“ von den begrenzten Identifler-Handling-Fähigkeiten dieser Maschine herrührt. Die Sprache wird mit der so entstandenen doppeldeutigen Benennung eher noch besser charakterisiert.

2 *postfix*: vgl. engl. *‚affixed after‘* (diese Schreibweise wird zuweilen auch als *klammerfreie Notation* bezeichnet)

3 *token*: vgl. engl. „Zeichen, Symbol“ (beispielsweise das ASCII ‚?‘,für den Print-Befehl in klassischen *BASIC*-Programmiersprachen)

4 Zahlreichen Forth-Systemen liegt ein Assembler bei, der bei Bedarf genutzt werden kann. Häufig sind diese Assembler selbst in Forth programmiert und fügen sich nahtlos in das System ein.

5 Forth-Slang: engl. *‚Outer Interpreter‘* (für Äußerer Interpreter)

6 sprich: engl. *‚colon‘*

7 Forth übersetzt seine Quellcodes ohne Compilerfehler. Dabei übersetzt es exakt das, was der Programmierer im Quellcode vorgibt. Bei Compilern anderer Programmiersprachen treten mit statitischer Wahrscheinlichkeit Fehler auf, die in der Arbeitsweise und der Programmierung eines dieser Compiler zu suchen sind, und zwar auch dann, wenn der zu übersetzende Quellcode in sich fehlerfrei und stimmig ist. Von einem solchen Verhalten sind Forth-Compiler in der Regel nicht betroffen.

8 Typische Forth-Programme haben einen Zeit-Overhead von 50 bis 200% gegenüber optimiertem Maschinen-Code. Optimierende Forth-Compiler reduzieren diesen Zeit-Overhead bis nahe zu null.

9 T. J. Zimmer: „*Tiny Pascal in Fig-Forth*“ (1981)

10 R. G. Loeliger: „*Threaded Interpretive Languages*“ (1981)

Quelle: „*Die Programmiersprache FORTH*“ von *Ronald Zech*, erschienen 1984 ff. im *Franzis*-Verlag (München), der hiesige Auszug ist eine Überarbeitung (2019)

Hauptprogramm ist

